Unification Diagnosis using Simulation Slices*

Venkatesh Choppella and Christopher T. Haynes
Computer Science Department
Indiana University
Bloomington, IN 47405, USA
{choppell,chaynes}@cs.indiana.edu

Abstract. We present a system for automatic and interactive generation of slicing information for diagnosing unification closure and non-unifiability. The solution term of each variable in a unifiable system of equations is witnessed by a slice of the original system relevant in deducing the solution. In case of failure, a unification slice simulating the failure is generated. The slicing information is built using the path-based framework proposed earlier by the authors[3,4]. The slicing information is obtained efficiently, and it is amenable to efficient optimization leading to slices that are typically minimally adequate to simulate a given unification.

1 Introduction

The process of term unification is at the heart of logic-based programming languages, deductive databases, automated deduction, theorem proving and and artificial-intelligence systems. Successful debugging of these systems, therefore, rests on a proper framework for the diagnosis of unification.

A framework based on *unification paths* was proposed in the first author's PhD thesis[3]. The objects of debugging in this framework are paths in the unification graph witnessing membership in the unification closure of the unification problem instance. These paths have a precise formal language semantics: the labels of these paths are suffixes of sentences of balanced parentheses (the semi-Dyck sets). Moreover, these paths may be used to construct proofs in the Logic of Unification[15]. Based on this analysis, the authors recently proposed an extension of the standard unification algorithm to compute unification paths[4].

In this paper, we present an interactive system that automatically generates *unification slicing information* using the framework proposed in [3,4]. Paths computed by the unification algorithm are used to construct linear unification simulation slices. Unification slices are unification subsystems simulating the membership of a single element in the unification closure, which corresponds to a symptom of unification failure in a non-unifiable system. Linear unification slices are unification subsystems in which vertex-sharing information is kept to a minimum and the non-empty subterms form a path.

* Under review for presentation at *ICLP '03: Nineteenth International Conference on Logic Programming*, December 9th-13th, 2003, Mumbai, India.

Our framework for source-tracking rests on a fundamental property relating paths in labeled directed graphs to paths in their quotient graphs. The labeled directed graph underlying a unification graph is obtained by orienting and labeling each edge of the unification graph. The rules for unification, consisting of equivalence and downward closure may be viewed as constructing a connectivity relation on the vertices of the underlying labeled directed graph.

The algorithm implementing the unification diagnosis framework is based on the following extension to the "standard" unification algorithm: for each call $unify(u, v)$ unifying two vertices u and v in the unification graph, we supply a *path expression* witnessing the connectivity of u and v in the unification graph. The computation and simplification of this path expression is achieved with a relatively low overhead.

Our framework provides source-tracking capability directly to the underlying unification engine. It may be used to build a debugging capability from the bottom up into a logic programming system.

Our approach may be considered as a trace-based approach. However, traditional trace-based approaches [7,8] focus on journaling sequences of events ordered by time and include recording, among other things, every binding made by the unification algorithm during the deduction process. Our framework, on the other hand, can be thought as computing traces with semantic content: each trace is a path in the semi-Dyck connectivity space of the unification graph rather than the temporal space of the unification algorithm's execution life-time. Furthermore, each trace encodes a deduction in the logic of unification. By exploiting algebraic properties of these encodings, our framework allows, through rewriting, considerable optimization in the size of these deductions.

The rest of this paper is divided into the following sections: Section 2 illustrates capabilities of our interactive unification diagnsosis system using an elaborate example. It also informally introduces some of the ideas defined later in the paper. Section 3 provides a quick introduction to the semantic principles underlying our system and formalizes the notion of unification slices. These slices are obtained from the path-based unification algorithm proposed in [4] and recalled in Section 4. Section 5 discusses related work. Section 6 concludes with suggestions for future work.

2 Interactive Unification Diagnosis

We illustrate interaction with our diagnostic unifier using a few simple examples. The unifier presented here is defined as a set of functions in implemented in *Chez Scheme 6.0a*. It is available for download as a tar-gzipped file from

`http://www.cs.indiana.edu/hyplan/chaynes/unif.tgz`

The system allows the definition, navigation, and querying of the unification system in an interactive setting. We start our interaction by defining a simple instance of a unification problem at a Scheme prompt:

```
> (define u-sys1
```

```
(make-unif-system 'u-sys1
  (list
    (=? (f X Y) (f (a) (b)) 1)
    (=? Z       (h Y)       2))))
```

This defines a unification system consisting the term equations $f(X,Y) \stackrel{?}{=} f(a,b)$ and $Z \stackrel{?}{=} h(Y)$. Terms are written in prefix form. Term constructors, or *functors*, are denoted by lower case letters, while variables are denoted by capital letters. Term equations are numbered for later reference. The unification system is first converted into a unification graph `u-gr1`, which is the primary data structure used for representing systems of term equations.

```
> (define u-gr1 (unif-system->unif-graph u-sys1))
> u-gr1
(unif-graph
    :id u-sys1
    :strict-vars (X Y Z) :functor-vars (w1 w2 w3 w4 w5)
    :functor-eqns
     ((= w1 (f X Y)) (= w2 (f w4 w3))
      (= w3 (b))     (= w4 (a))       (= w5 (h Y))))
    :unif-eqns ((=? w1 w2 1) (=? Z w5 2)))
```

The unification graph is defined as a system of *flat term equations*. The flat system makes the sharing structure implicit in unification systems explicit. For each occurrence of a non-variable term there is a *functor variable*. Functor variables are denoted by lowercase symbols w_1, w_2, etc. Each variable of the unification system (X,Y, etc.) is mapped to a *strict* variable of the same name. *Functor equations* map each functor variable to a flat term. Each term equation in the unification system is mapped to a *unification equation* between the unification graph variables.

Now we are ready to run the unification algorithm on `u-gr1`.

```
> (unify u-gr1)
unified
```

This indicates that the unification graph has been unified. The unification process results in the computation of the unification closure of the graph, which is the smallest downward-closed equivalence on the graph vertices containing the unification equations. In our example this closure is the equivalence

$$\{\{X, w_4\}, \{Y, w_3\}, \{w_1, w_2\}, \{Z, w_5\}\}$$

Now the unification graph is ready for examination. We first examine the solution of variables:

```
> (sol-term X)
(a)
```

```
> (sol-slice X)
((=? (f X _) (f (a) _) 1))

> (sol-term w4)
(a)

> (sol-term Z)
(h (b))
> (sol-slice Z)
(
  (=? (f _ Y) (f _ (b)) 1)
  (=? Z       (h Y))    2)
)
```

For each functor and strict variable, `sol-term` returns the solution term computed by the unification. The function `sol-slice` returns a *slice* of the original system of equations. A slice is a *weakening* of the original system of term equations obtained by term erasure, which involves replacing some subterms of the original system by distinct anonymous variables each denoted by the `_` symbol. In the example above, the slices erase information that is irrelevant to a particular unification inference. For example, the solution a of X is obtained by the slice $f(X, \_) \stackrel{?}{=} f(a, \_)$ of the original term equation $f(X, Y) \stackrel{?}{=} f(a, b)$ numbered 1. Any further erasure of information from the slices fails to derive the solution dictated by the original system of unification. For instance, by weakening the second slice in the solution of Z to $Z \stackrel{?}{=} h(\_)$, we no longer can derive the solution of $Z = h(b)$ from the resultant set of slices. Thus, the slicing information generated in this case is *minimally sufficient* to simulate the solutions of variables from the original unification graph.

2.1 Paths witness unification closure membership

The path-based framework for unification diagnosis views unification as the process of constructing special connectivity relations between vertices in the graph. For every element (u, v) in the unification closure, there is a *unification path* from u to v in the unification graph.

```
> (unifies? X w4)
(-w1.f_0 +1 +w2.f_0)
```

The query `(unifies? X w4)` returns a path from the vertex `X` to `w4` in the original graph with the assumption that edges may be traversed in either direction. Traversal of an edge `e` is denoted `+e` or `-e`, depending on whether the traversal is parallel or opposite to the edge's orientation. Thus the expression `(-w1.f_0 +1 +w2.f_0)` denotes the path consisting of three edges: The first edge `-w1.f_0` connects the vertex `X`, which is the zero'th child of the vertex `w1` labeled `f`, to `w1`. The edge `+1` denotes the (positive) traversal of the edge corresponding to

the unification equation labeled 1, which connects `w1` to `w2`. Finally, the edge `+w2.f_0` denotes the positive traversal of the branch edge from `w2` labeled with the functor symbol `f` to its zero'th child `w4`. Using this path, the proof of $x = w_2$ may be constructed using the following chain of reasoning:

1. $f(X, \_) = w_1$ Branch Equation `-w1.f_0`
2. $w_1 = w_2$ Term Equation 1
3. $w_2 = f(w_4, \_)$ Branch Equation `+w2.f_0`

Note that these three primitive steps together correspond to the "subterm unification", or downward closure rule in unification, which states that if two terms unify, then their corresponding immediate subterms unify as well.

Note how the upward (-) and downward (+) traversals cancel each other out for paths representing membership in the unification closure. More precisely, each edge in the traversal has an associated *projection label*.

```
> (path->projection-labels (unifies? X w4))
(-f_0 *e* +f_0)
```

The projection label of a branch edge from a functor vertex whose functor is f to its ith child is denoted f_i. The special projection label `*e*` denotes the empty label ϵ. The projection labels concatenated together give the string $f_0^{-1} f_0$, which, after 1-sided cancellation, yields ϵ.

The quotient graph induced by the closure relation is a term graph (consisting of strict and functor vertices and functor equations, but not unification equations). The vertices of this quotient graph are the equivalence classes of the closure relation. Furthermore, for every branch edge from u to v in the original graph, there is branch edge $[u]$ to $[v]$ in the quotient term graph. This connectivity in the quotient graph, however is due to connectivity in original graph witnessed by a path. Consider the path from the vertex `Z` to `w3`:

```
> (reaches? Z w3))
(+2 +w5.h_0 -w1.f_1 +1 +w2.f_1)
> (path->projection-labels (reaches? Z w3))
(*e* +h_0 -f_1 *e* +f_1)
```

The "net" traversal of this path is obtained by reducing the string $\epsilon h_0 f_0^{-1} \epsilon f_1$ using 1-sided cancellation, which yields h_0. This implies that the vertex $[w_3]$, the equivalence containing w_3 is the zeroth child of $[Z]$ in the quotient term graph.

2.2 Unification slices from paths

Obtaining unification slices from a given path involves parsing the path and dividing it into a sequence of segments. Each segment is a triple consisting of a (possibly empty) negative, or "upward" traversal from a vertex to its ancestor, a unification equation, followed by a positive, or "downward" traversal from a vertex to its descendent. Each segment corresponds to a slice of a unification equation in the original system.

```
> (path->segments (reaches? Z w3))
((() +2 (+w5.h_0)) ((-w1.f_1) +1 (+w2.f_1)))
```

The first segment yields the slice $Z \stackrel{?}{=} h(Y)$; the second segment yields $f(\_, Y) \stackrel{?}{=} f(\_, b)$. Note that given a term, a path in that term defines a term slice which is a weakening of the original term in which all subterms not along the given path are empty. Each segment identifies two term slices linked together with a term equation number, as computed by the `path->slices` function:

```
> (path->slices  (reaches? Z w3))
((=? Z (h Y) +2) (=? (f _ Y) (f _ (b)) +1))
```

2.3 Slices for unification failure

Unification failure is a special condition when the unification closure contains a pair of functor vertices with different labels (CLASH), or the quotient term graph contains a cycle (CYCLE).

Consider the single-equation system

```
> (define clash-sys
  (make-unif-system 'clash-sys
    (list
      (=? (f V V)  (f (a) (b)) 1))))

> (define clash-gr (unif-system->unif-graph clash-sys))
> clash-gr
(unif-graph
    :id  clash-sys
    :strict-vars (V) :functor-vars (w12 w13 w14 w15)
    :functor-eqns
    ((= w12 (f V V)) (= w13 (f w15 w14))
     (= w14 (b))     (= w15 (a)))
    :unif-eqns ((=? w12 w13 1)))
```

Unification of the above equation causes a clash to be reported. The report produces the symptom of the clash, a path witnessing this clash, and a unification slice derived from this path.

```
> (run clash-gr)
UNIFICATION failed!
CLASH between vertex w15 labeled a AND vertex w14 labeled b
WHILE unifying equation (=? w12 w13 1):

derived from the PATH
 (-w13.f0 -1 +w12.f0 -w12.f1 +1 +w13.f1)
corresponding to the unification  SLICE
 ((=? (f (a) _) (f V _)   -1)
  (=? (f _ V)   (f _ (b)) +1))
```

The error announces that a path can be traced from the vertex w_{15} labeled a to the vertex w_{14} labeled b. The path yields a unification slice consisting of two term equation slices. The first term equation connects the vertex labeled a to V, while the second connects V to the vertex labeled b. The separation of the two term equation slices de-emphasizes the role of the term $f(V,V)$ in which the vertex V is obtained by two different projections of the *same* functor variable. The separate term equations suggest that the two occurrences of the variable V can be obtained as different projections of *possibly different* functor variables. In other words, the set of terms $\{f(V,\_), f(\_,V)\}$ is *weaker* than the singleton-set $\{f(V,V)\}$. The underlying notion here is that of an ordering made precise later in the paper.

3 Unification paths

In this section we briefly review the graph-theoretic context for understanding the semantics of the the information computed by our unification diagnosis algorithm. The interested reader is referred to [3,4] for further details, including proofs of the main results.

Given a signature Σ, a family of Σ-terms over X is represented as a *term graph* $T = \langle W, X, b\rangle$, where X is the set of *strict* vertices, W is a set of *functor vertices*, and $b : W \longrightarrow \Sigma(W \cup X)$ is the *immediate subterm function*. If $b(w) = f(u_1, \ldots, u_n)$, then, we say that the *functor label* of w is f and that for each i, u_i is the ith child of w. The term graph representation explicates the assumption that variable vertices are shared and functor vertices *may* be shared. A *unification graph* is a pair $\langle T, E\rangle$ where $T = \langle W, X, b\rangle$ is a term graph and E a relation over $W \cup X$, is a set of *unification equations*.

The basic object of our formal analysis is the labeled directed graph (LDG) underlying a unification graph. A labeled directed graph $G = \langle \Sigma, V, D\rangle$ consists of an alphabet Σ, a set of vertices V and a set of labeled directed edges $u \xrightarrow{\delta} v \in D$, where $u, v \in V$ and $\delta \in \Sigma$.

Given a *signature* Σ, an alphabet in which each symbol has an *arity*, $\Sigma_{\mathbf{N}} = \{f_i | f \in \Sigma, 1 \leq i \leq \mathit{arity}(f)\}$ is called the *projection alphabet of* Σ. The labeled directed graph *underlying* a unification graph $G = \langle W, X, b, E\rangle$ is the LDG $\langle \Sigma_{\mathbf{N}} \cup \{\epsilon\}, W \cup X, D\rangle$, where, the *branch edge* $u \xrightarrow{f_i} v$ is in D if v is the ith child of the functor vertex u labeled f in G, and the *equational edge* $u \xrightarrow{\epsilon} v$ is in D if $(u, v) \in E$. For convenience, we identify unification graphs with their underlying LDG's. The rest of our formalism works directly with LDG's. Given an LDG $G = \langle \Sigma, V, D\rangle$, a relation R on vertices V is *downward closed* if for each uRv, $u \xrightarrow{\delta} u' \in D$ and $v \xrightarrow{\delta} v' \in D$ implies that $u'Rv'$. The *unification closure* $\sim$ of an LDG G is the least downward-closed equivalence relation on the vertices of G containing the equational edges of G. The closure $\sim$ on G induces a quotient graph $G/{\sim} = \langle \Sigma, V/{\sim}, D/{\sim}\rangle$, where $V/{\sim}$ consists of the equivalence classes of $\sim$ and if $[u], [v]$ are equivalence classes containing vertices u and v of G respectively, $[u] \xrightarrow{\delta} [v] \in D/{\sim}$ if and only if $u \xrightarrow{\delta} v \in D$.

A fundamental result due to Paterson and Wegman[19] relates the unifiability of a graph to properties of its quotient graph: a graph G is unifiable if and only if the quotient graph $G/\sim$ is *acyclic*, and *homogeneous*, that is, for any two functor vertices $w \sim w' \in G$, the functor labels of w and w' are identical.

In order to debug unification, however, it is necessary to formulate the non-unifiability of a unification graph G directly in terms of its connectivity properties, without considering its quotient graph. The key to this formulation is to consider the labels of paths obtained by a traversal of the unification graph edges in both parallel (positive) and anti-parallel (inverse) directions. An inverse traversal accumulates a "inverse" label. In order to formalize this, we first define the inverse of an alphabet Σ, denoted Σ^{-1} , which is obtained obtained by mapping every symbol s of Σ to the symbol s^{-1}. Given a labeled directed graph $G = \langle \Sigma, V, D \rangle$, its *inverse* G^{-1} is the graph $\langle \Sigma^{-1}, V, D^{-1} \rangle$ obtained by reversing the edges (and their labels) of D with the constraint that $\epsilon^{-1} = \epsilon$.

If each symbol of Σ is considered a closed parentheses symbol whose matching open parenthesis symbol is in Σ^{-1}, then we write $D^0(\Sigma)$ to denote the language of balanced sentences over the parentheses symbols $\Sigma^{-1} \cup \Sigma$. In formal language theory, this language is called the *semi-Dyck set* over Σ. Let $D^*(\Sigma)$ denote the language of suffix language of $D^0(\Sigma)$. Semi-Dyck sets have nice cancellative properties: The *one-sided cancellation* rules $\mathbf{D}(\Sigma)$ consist of rewriting a substring $s^{-1}s$ to ϵ. Thus, for a sentence l over $\Sigma \cup \Sigma^{-1}$ the *normal form* $\mu_{\mathbf{D}(\mathbf{\Sigma})}$ is the unique sentence obtained by applying one-sided cancellation. It is easy to see that the suffix language $D^*(\Sigma)$ of Dyck sets is exactly the set of of all sentences l over $\Sigma \cup \Sigma^{-1}$ such that $\mu_{\mathbf{D}(\mathbf{\Sigma})}(l) \in \Sigma^*$. As a special case, the set of well-balanced sentences are those whose normal form is ϵ.

A *unification path from u to v over G* is a path p in $G \cup G^{-1}$ such that the label $l(p)$ is a sentence in the language $D^*(\Sigma)$. The set of unification paths from vertex u to v whose label is in $D^0(\Sigma)$ is denoted P^0_{uv}. Unification paths over G completely characterize connectivity in the quotient graph $G/\sim$. This is the essence of unification "source-tracking" because we can track paths in the quotient graph in terms of their "source" paths in the original graph. We write $G \models u \xrightarrow{l} v$ to denote that there is a path labeled l from vertex u to vertex v in G.

Theorem 1. *(Soundness and Completeness of unification paths) [3]*
Let G be an LDG $\langle \Sigma, V, D \rangle$ whose unification closure is $\sim$. If $G \cup G^{-1} \models u \xrightarrow{l} v$ and $l \in D^*(\Sigma)$, then $G/\sim \models [u]_\sim \xrightarrow{\mu_{\mathbf{D}(\Sigma)}(l)} [v]_\sim$.
Let $G = \langle \Sigma, V, D \rangle$ be an LDG with unification closure $\sim$. If $G/\sim \models [u]_\sim \xrightarrow{l'} [v]_\sim$, then $G \cup G^{-1} \models u \xrightarrow{l} v$ for some $l \in D^*(\Sigma)$ such that $\mu_{\mathbf{D}(\Sigma)}(l) = l'$.

3.1 Term and Equation Slices

Term and Unification slices are defined by considering Σ-terms over a set of variables containing the distinguished variable "_". We refer to terms over this

extended variable set as *term slices*. We define the following *weakening* ordering over terms slices: If V is a set of variables, and Σ is a signature, the set of *Σ-term slices* over V is the set of Σ-terms over $V \cup \_$, where $\_$ denotes a special variable not occuring in V. The set $V \cup \_$ may be considered a flat poset, with the ordering $x \sqsubseteq y$ if and only if $x = y$ or $x = \_$.

The set of *term equations slices* is the set of term equations over term slices. The flat poset $\sqsubseteq$ induces a partial order $\sqsubseteq$ on term slices and term equation slices:

1. $x \sqsubseteq y$, if $x, y \in V \cup \_$ and $x = y$ or $x = \_$.
2. $f(\tau_1, \ldots, \tau_n) \sqsubseteq f(\tau'_1, \ldots, \tau'_n)$, if $\tau_i, \tau'_i \in T_\Sigma(V \cup \_)$ and $\tau_i \sqsubseteq \tau'_i$, $1 \leq i \leq n$.
3. $\tau_1 \stackrel{?}{=} \tau_2 \sqsubseteq \tau'_1 \stackrel{?}{=} \tau'_2$, if $\tau_1, \tau_2, \tau'_1, \tau'_2 \in T_\Sigma(V \cup \_)$, $\tau_1 \sqsubseteq \tau'_1$ and $\tau_2 \sqsubseteq \tau'_2$.

The partial order $\sqsubseteq$ on weak term equations induces a lower power-domain order over sets of weak term equations: $E \sqsubseteq E'$ if for each $\eta \in E$, there is an $\eta' \in E'$ such that $\eta \sqsubseteq \eta'$. We say that E' is a *unification slice of* E. Given a set E of weak term equations, the set of *unification slices of* E is defined to be the downset $E{\downarrow}$ of E. Weakenings are a generalization of *linearizations* (Le Chenadec [14]), which are obtained by replacing any one occurrence of a variable occurring more than once with a $\_$.

A *linear term slice* is a term slice in which the non-empty terms are along a path from the root of the term slice. Formally, the set of *linear Σ-term slices* over variables V is the least set such that

1. every element of $V \cup \_$ is a linear Σ-term slice.
2. if τ is a linear term slice, and $f \in \Sigma$ such that $\mathit{arity}(f) = n$, then $f(\__1, \ldots, \tau, \ldots, \__n)$ is a linear term slice.

A *linear term equation slice* is a term equation slice whose left and right hand side term slices are linear.

4 Implementation of unification path generation

In this section we recall the unification algorithm proposed in [3,4]. The algorithm for unification path generation generates *unification path expressions*, which reduce to unification paths. To illustrate the integration of computation of unification paths, we consider the quadratic-time unification algorithm of Corbin and Bidoit[5] as presented in (Baader and Siekmann [1]). Other unification algorithms may be used as well. The resulting algorithm is shown in Figure 1. The binding field of each variable and the return value of *find* is a tuple containing two objects: a pointer to the root of an equivalence class and a unification path expression denoting the path expression to root. The procedures *unify* and *union* also carry an extra parameter that is a unification path expression. The procedure *occurs?*(u, v) returns either no or $\mathsf{yes}(x, p)$, where x is either 0 or $+$, and p is a unification path expression from u to v in the unification graph containing the vertices u and v.

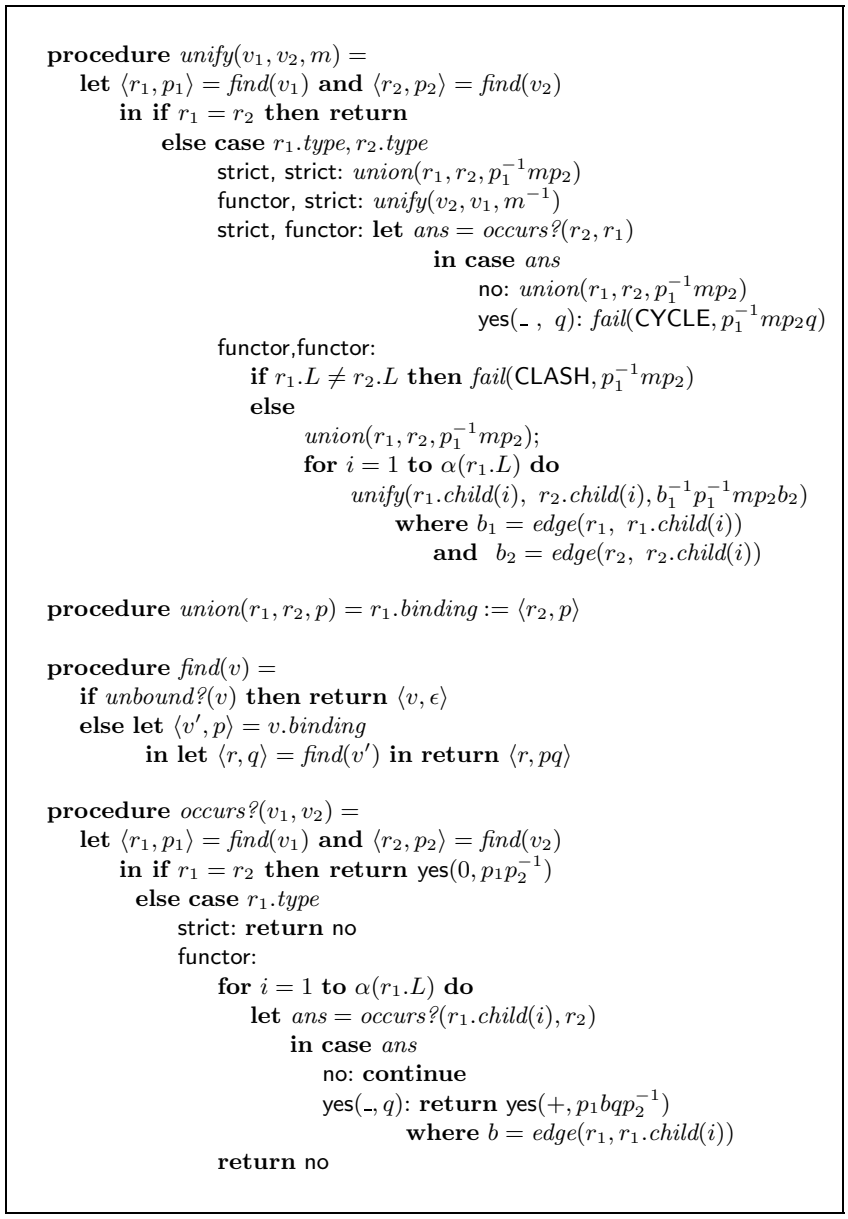

Fig. 1. Unification Algorithm with unification path expression generation

If $m : \tau_1 \stackrel{?}{=} \tau_2$ is a term equation, its unification graph G contains term trees rooted at vertices u_1 and u_2, representing the terms τ_1 and τ_2 respectively, and an equational edge from v_1 to v_2 labeled m. The first call to unify is *unify*(u_1, u_2, m). The unification algorithm maintains the invariant that for every call *unify*(v_1, v_2, p), $p \in P^0_{v_1 v_2}$. Also, if *find*$(v_1) = \langle r_1, p_1 \rangle$, then $p_1 \in P^0_{v_1 r_1}$. Similarly, if *find*$(v_2) = \langle r_2, p_2 \rangle$, then $p_2 \in P^0_{v_2 r_2}$. The call *union*(r_1, r_2, p) maintains the invariant that $p \in P^0_{r_1 r_2}$. If the call *unify*(v_1, v_2, p) results in the call *union*$(r_1, r_2, (p_1)^{-1} p p_2)$, binding r_1 to r_2, where *find*$(v_1) = \langle r_1, p_1 \rangle$ and *find*$(v_2) = \langle r_2, p_2 \rangle$, then the vertex r_2 and the path expression $(p_1)^{-1} p p_2$ are stored as the binding information $r_1.binding$. The path expression $(p_1)^{-1} p p_2$ implicitly carries the information that v_1 is connected to v_2 with the path p. To recover the information that v_1 is connected to v_2 by p, we simply compute the path expression following the *binding* pointers from v_1 to v_2, which is done in three steps:

1. compute the path expression from v_1 to r_1 using *find*(v_1), which is p_1.
2. compute the path from r_1 to r_2 using *find*(r_1), which is $(p_1)^{-1} p p_2$.
3. compute the path from r_2 to v_2, which is $(p_2)^{-1}$.

It turns out that the path expressions computed by the unification algorithms may be considered as elements of the group generated by the edges of the unification graph. This allows us to use the group simplification rules to simplify path expressions. Using these rules, the concatenation of the above three path expressions, $p_1 (p_1)^{-1} p p_2 (p_2)^{-1}$, simplifies to p, which normalizes using group rewriting to a unification path. The details are in[3].

4.1 Efficiency

The cost of constructing unification path expressions at each point in the algorithm is constant time per call to *unify*, *find*, *union* and *occurs?*, assuming paths are represented as terms sharing structure. Thus the addition of source-tracking to the unification algorithm increases runtime by only a constant factor.

Since normalization is orthogonal to the building of path expressions, it may be performed once the unification path expression has been computed. It is easily seen that normalization using group rewriting takes time proportional to the size of the term being normalized. In practice, we have observed that group rewriting results in a path whose size is significantly smaller.

5 Related Research

5.1 Logic programming and unification-based systems

The main focus in debugging of logic programming is on building declarative debugging frameworks[21,18,17,11], trace-based[7,8] and annotation-based formalisms[13], though program slicing has received somewhat marginal attention, as noted in[9]. However, early work in slicing-based unification diagnosis was

carried out by Port [20]. His approach was guided by the goal to perform unification failure analysis by identifying minimally unifiable subsets. Port's algorithm attempts to construct regular path expressions over the unification graph. Our work shows that the path expressions of relevance are context-free, not regular.

Cox [6] and Chen et al. [2] propose an algorithm to derive maximally unifiable subsets and minimally non-unifiable subsets of term equations employed as the basis for developing search strategies for breadth-first resolution of logic programs. Our extension to the unification algorithm keeps track of information that is more precise than subsets. Furthermore, the information maintained statically in the auxiliary graph constructed by the algorithm of Chen et al. can be generated dynamically in our extension to the unification algorithm.

5.2 Unification algorithms in type inference

Wand [23] modified the unification algorithm to accumulate reasons when traversing the the unification graph. Unfortunately, Wand did not show how the reasons together simulate the error. More importantly, Wand's algorithm sometimes fails to report reasons critical to the reconstruction of the unification failure, and at other times returns much redundant information. Eliminating redundant inferences from reason lists in a systematic way can be done by abandoning the algorithm's set-based approach, and instead using the path-based approach suggested by our algorithm.

Johnson and Walz [12,22] introduce "error-tolerant" unification, in which a multi-set of type constraints is solved by using a disjunction of constraints rather than a conjunction. Their scheme rests on a complicated algorithm that derives implied type constraints obtained from the original type constraints by applying the rules of substitution and transitivity. Unfortunately, their work presents no correctness and completeness criterion against which their algorithm can be judged.

The attribute grammar approach of Johnson and Walz and others confuses the generation of type constraints, which is directed by the syntax of the program, with the solution of these constraints, which is directed by the geometry of the constraints themselves. In any unification-based type system, types are more accurately viewed as flowing along special paths of the unification graph, not through the source code. Other approaches include interactive explanation-based systems[10], but these lack automation because the user is expected to navigate the unification graph.

Because these and other more recent diagnosis algorithms[16,24] are built on *top* of unification, they are unable to remove redundant inferences introduced by the underlying unification process.

6 Conclusions and Future Work

We have demonstrated a system for automatic, efficient extraction of unification slicing information for the diagnosis of unification. While our slicing information

has a semantic basis, it is also intuitive because it is based on the idea of path connectivity in the unification graph. We are currently working on providing a graphical front-end to our system to help visualize unification paths. The main direction for future work, however, lies in integrating our low-level mechanism into existing debugging technology for languages like Prolog and other similar unification-based logic programming engines.

References

1. Baader, F., and Siekmann, J. Unification theory. In *Handbook of Logic in Artificial Intelligence and Logic Programmaning*, D. M. Gabbay, C. J. Hogger, and J. A. Robinson, Eds. Oxford University Press, 1993.
2. Chen, T. Y., Lassez, J.-L., and Port, G. S. Maximal unifiable subsets and minimal non-unifiable subsets. *New Generation Computing* (1986), 133–152.
3. Choppella, V. *Unification Source-tracking with Application to Diagnosis of Type Inference*. PhD thesis, Indiana University, August 2002. IUCS Tech Report TR566.
4. Choppella, V., and Haynes, C. T. Source-tracking unification. To be presented at the 19th International Conference on Automated Deduction, CADE-19, Miami, USA, August 2003.
5. Corbin, J., and Bidoit, M. A rehabilitation of robinson's unification algorithm. In *Information Processing* (1983), R. E. A. Mason, Ed., Elsevier Science Publishers (North Holland), pp. 909–914.
6. Cox, P. T. Finding backtrack points for intelligent backtracking. In *Prolog Implementation*, J. Campbell, Ed. 1984, pp. 216–233.
7. Ducasse, M. Abstract views of prolog executions in Opium. In *Proceedings of the 1991 International Symposium on Logic Programming* (San Diego, California, Oct. 1991), pp. 18–34.
8. Ducassé, M. Opium: An extensible trace analyzer for Prolog. *Journal of Logic Programming 39* (1999), 177–223.
9. Ducassé, M., and Noyé, J. Logic programming environments: Dynamic program analysis and debugging. *The Journal of Logic Programming 19*, 20 (1994), 351–384.
10. Duggan, D., and Bent, F. Explaining type inference. *Science of Computer Programming 27*, 1 (July 1996), 37–83.
11. Ferrand, G. Error diagnosis in logic programming, an adaptation of E.Y. Shapiro's method. *Journal of Logic Programming 4*, 3 (Sept. 1987), 177–198.
12. Johnson, G. F., and Walz, J. A. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Proceedings of the 13th ACM Symposium on Programming Languages* (1986), pp. 44–57.
13. Kulas, M. Debugging Prolog using annotations. *Electronic Notes in Theoretical Computer Science 30*, 4 (2000). `http://www.elsevier.nl/locate/entcs/volume30.html`.
14. Le Chenadec, P. On positive occur-checks in unification. Tech. Rep. 792, INRIA, January 1988.
15. Le Chenadec, P. On the logic of unification. *Journal of Symbolic computation 8*, 1 (July 1989), 141–199.
16. Lee, O., and Yi, K. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages 20*, 4 (July 1998), 707–723.
17. Lloyd, J. Declarative error diagnosis. *New Generation Computing 5*, 2 (1987), 133–154.

18. Naish, L. A declarative debugging scheme. *Journal of Functional and Logic Programming 1997*, 3 (April 1997).
19. Paterson, M., and Wegman, M. Linear unification. *J. Comput. Syst. Sci. 16*, 2 (1978), 158–167.
20. Port, G. S. A simple approach to finding the cause of non-unifiability. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium* (1988), R. A. Kowalski and K. A. Bowen, Eds., MIT Press, pp. 651–665.
21. Shapiro, E. Y. *Algorithmic program debugging.* MIT Press, Cambridge, Massachusetts, 1983.
22. Walz, J. A. *Extending Attribute Grammars and Type Inference Algorithms.* PhD thesis, Cornell University, February 1989. TR 89-968.
23. Wand, M. Finding the source of type errors. In *13th Annual ACM Symp. on Principles of Prog. Languages.* (January 1986), pp. 38–43.
24. Yang, J., Trinder, P., Michaelson, G., and Wells, J. Improved type error reporting. In *Proceeding of Implementation of Functional Languages, 12th International Workshop* (September 2000), pp. 71–86.